بسم‌الله‌الرحمن‌الرحیم

درخت های قرمز-سیاه - Trees Red-Black

مرداد ۱۴۰۴

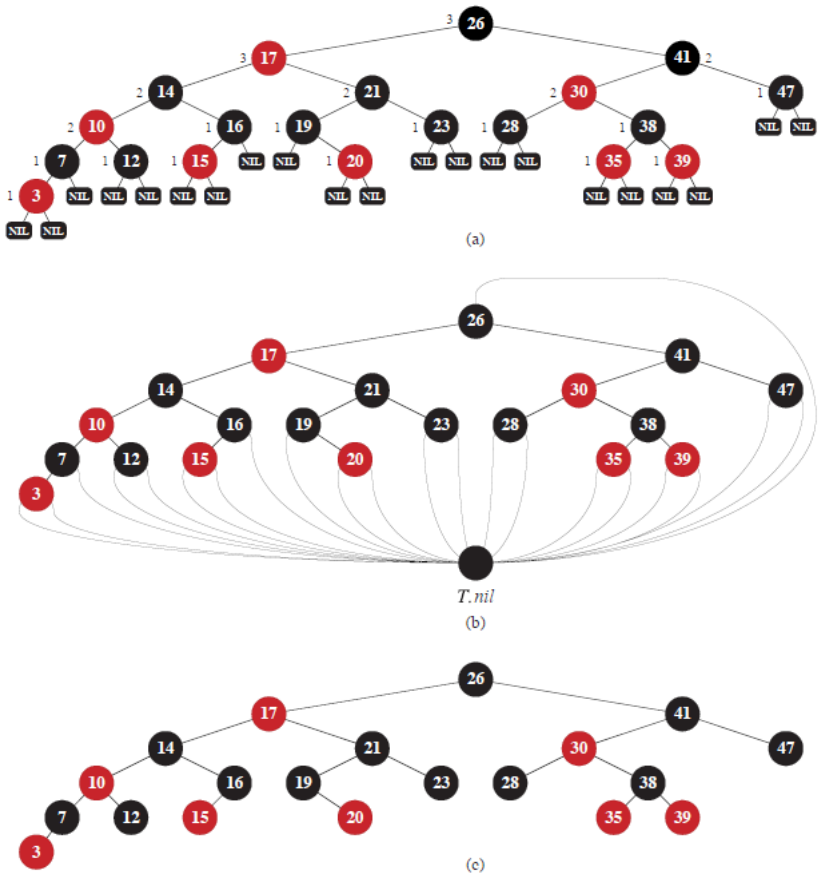
فهرست مطالب

۱ مقدمه

Red-black و یا درخت های قرمز و سیاه درخت های جستجوی دودویی هستند که هر گره آنها دارای رنگ قرمز و یا مشکی است. درخت red-black دارای ارتفاع $\log n$ است ، بنابراین عملیات‌های درج، جستجو و حذف در آن در بدترین حالت $O(\log n)$ زمان می‌برند.

۲ تعریف درخت قرمز-سیاه

درخت قرمز-سیاه نوعی Binary Search Tree است که در آن به هر گره یک رنگ (قرمز یا سیاه) نسبت داده می‌شود. قوانین رنگ‌بندی و چرخش‌ها باعث حفظ توازن درخت در حین عملیات درج و حذف می‌شوند.



شکل ۱: نمونه تصویر درخت سیاه-قرمز x

۳ ویژگی‌های درخت قرمز-سیاه

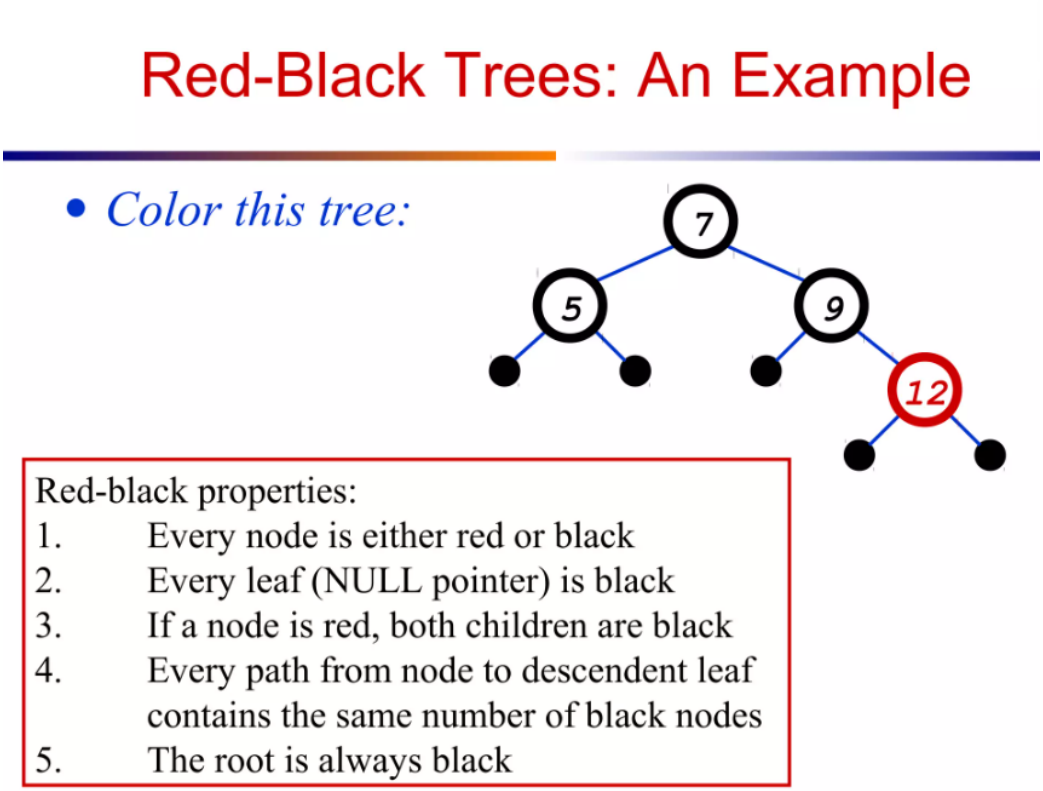
۱. هر گره یا قرمز است یا سیاه.

۲. ریشه همیشه سیاه است.

۳. همه‌ی برگ‌ها (گره‌های NIL یا تهی) سیاه هستند.

۴. اگر گره‌ای قرمز باشد، فرزندان آن حتماً سیاه هستند.

۵. تعداد گره‌های سیاه در هر مسیر از یک گره تا برگ‌هایش باید یکسان باشد.



شکل ۲: نمونه درخت قرمز-مشکی

۴ ارتفاع درخت قرمز-سیاه

حداکثر ارتفاع

- حداقل طول درخت Red-Black برابر با **black-height** یا ارتفاع گره‌های مشکی است.
- حداقل ارتفاع h برای n گره برابر است با:

$$h \geq \lfloor \log_2(n+1) \rfloor$$

این حالت زمانی رخ می‌دهد که درخت Red-Black به صورت یک درخت دودویی کامل باشد، یعنی:

$$n = 2^h - 1$$

- در بدترین حالت، ارتفاع Red-Black Tree از رابطه زیر پیروی می‌کند:

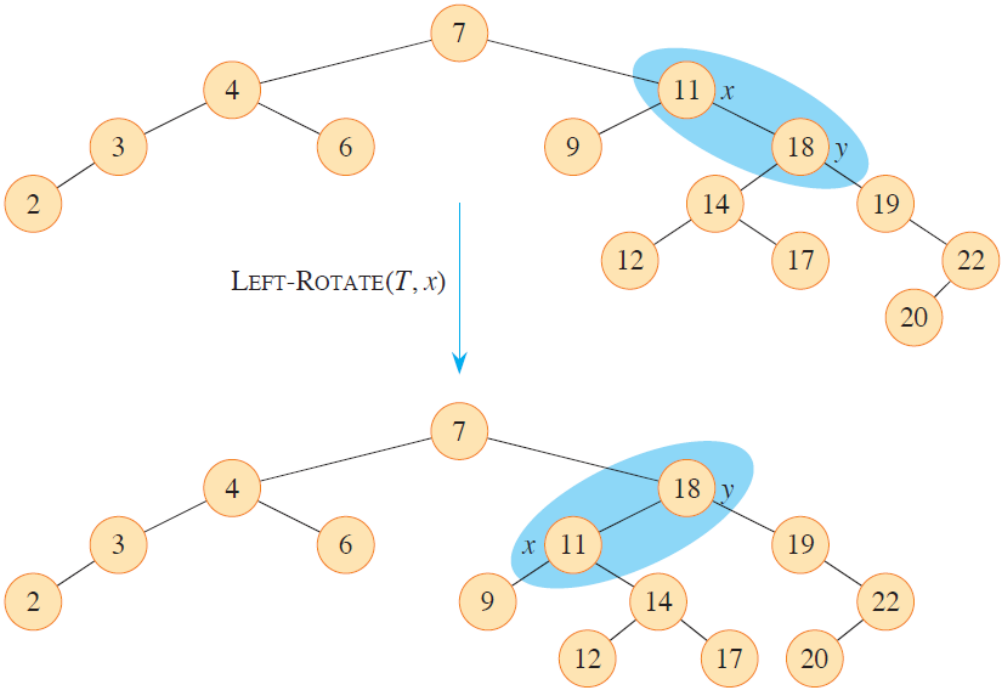
$$h \leq 2 \log_2(n+1)$$

۵ چرخش در درخت قرمز-سیاه

چرخش‌ها عملیات اصلی برای حفظ ویژگی‌های درخت قرمز-سیاه در حین درج و حذف هستند. این عملیات به صورت محلی ساختار درخت را تغییر می‌دهند اما خاصیت درخت جستجوی دودویی را حفظ می‌کنند و در زمان $O(1)$ انجام می‌شوند. دو نوع چرخش وجود دارد: چرخش به چپ و چرخش به راست.

LEFT-ROTATE ۱.۵

در چرخش به چپ حول گره x، فرض می‌شود که فرزند راست آن یعنی y تهی نیست. این چرخش باعث می‌شود y به جای x قرار گیرد و x به عنوان فرزند چپ y درآید. زیردرخت چپ y نیز به عنوان زیردرخت راست x متصل می‌شود.



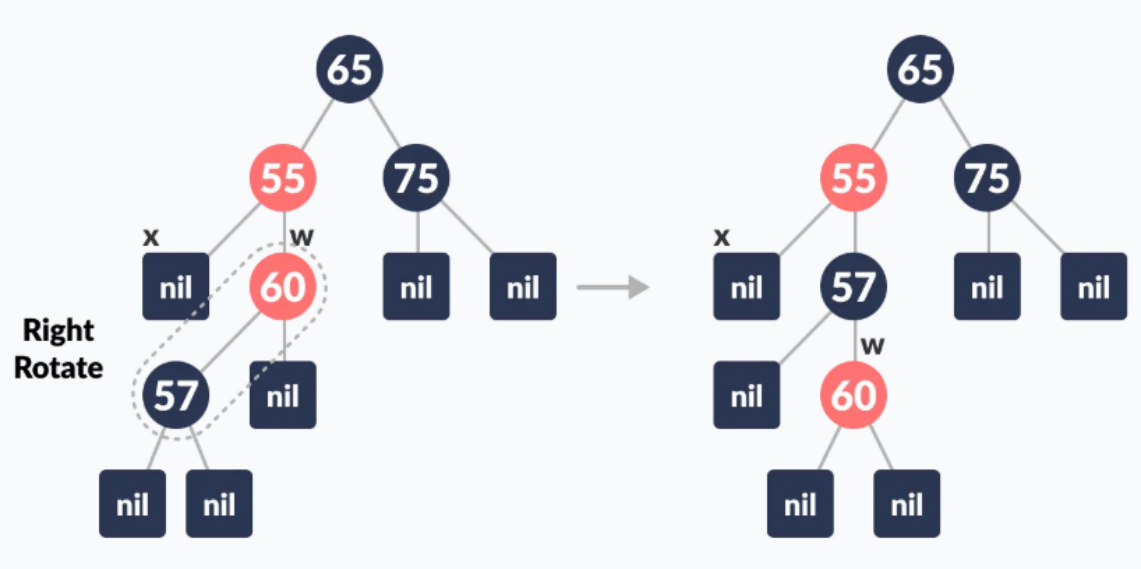
شکل ۳: عملکرد چرخش به چپ حول گره x

```
LEFT-ROTATE(T, x):
    y = x.right
    x.right = y.left
    if y.left != T.nil:
        y.left.p = x
    y.p = x.p
    if x.p == T.nil:
        T.root = y
    else if x == x.p.left:
        x.p.left = y
    else:
        x.p.right = y
    y.left = x
    x.p = y
```

Listing :۱ Left Rotation Pseudocode

RIGHT-ROTATE ۲.۵

چرخش به راست عملیات معکوس چرخش به چپ است. در این چرخش حول گره y، فرض می‌شود که فرزند چپ آن یعنی x تهی نیست.



شکل ۴: عملکرد چرخش به راست حول گره y=w

```
RIGHT-ROTATE(T, y):
  x = y.left
  y.left = x.right
  if x.right != T.nil:
    x.right.p = y
  x.p = y.p
  if y.p == T.nil:
    T.root = x
  else if y == y.p.right:
    y.p.right = x
  else:
    y.p.left = x
  x.right = y
  y.p = x
```

Listing :۲ Right Rotation Pseudocode

۶ درج در درخت قرمز-سیاه

درج گره جدید ممکن است باعث نقض قوانین درخت شود و ساختار آن را به هم بزند. در این شرایط، برای حفظ خاصیت‌های درخت، از چرخش‌ها و رنگ‌آمیزی مجدد استفاده می‌شود. این اصلاحات باعث می‌شوند که درخت به حالت متعادل و درست بازگردد. هدف اصلی این است که خاصیت‌های اساسی درخت مانند تعادل رنگ‌ها و ترتیب گره‌ها حفظ شود. بر اساس وضعیت درخت پس از درج گره، سه حالت مختلف ممکن است پیش بیاید. هر یک از این حالات نیازمند راه‌حل خاص خود است که با چرخش‌ها و تغییر رنگ‌ها اصلاح می‌شود. این فرآیندها به صورت خودکار و مرحله به مرحله انجام می‌گیرند. در ادامه، هر یک از این حالت‌ها به صورت دقیق و گام به گام بررسی خواهند شد.

کد الگوریتم درج

```
RB-Insert(T, z):
  y = NIL
  x = T.root
  while x != NIL:
    y = x
    if z.key < x.key:
    x = x.left
    else:
      x = x.right
  z.p = y
  if y == NIL:
    T.root = z
  else if z.key < y.key:
    y.left = z
  else:
    y.right = z
  z.left = NIL
  z.right = NIL
  z.color = RED
  RB-Insert-Fixup(T, z)
```

Listing :۳ Red-Black Tree Insertion

حالت ۱: عموی گره جدید (y) قرمز است

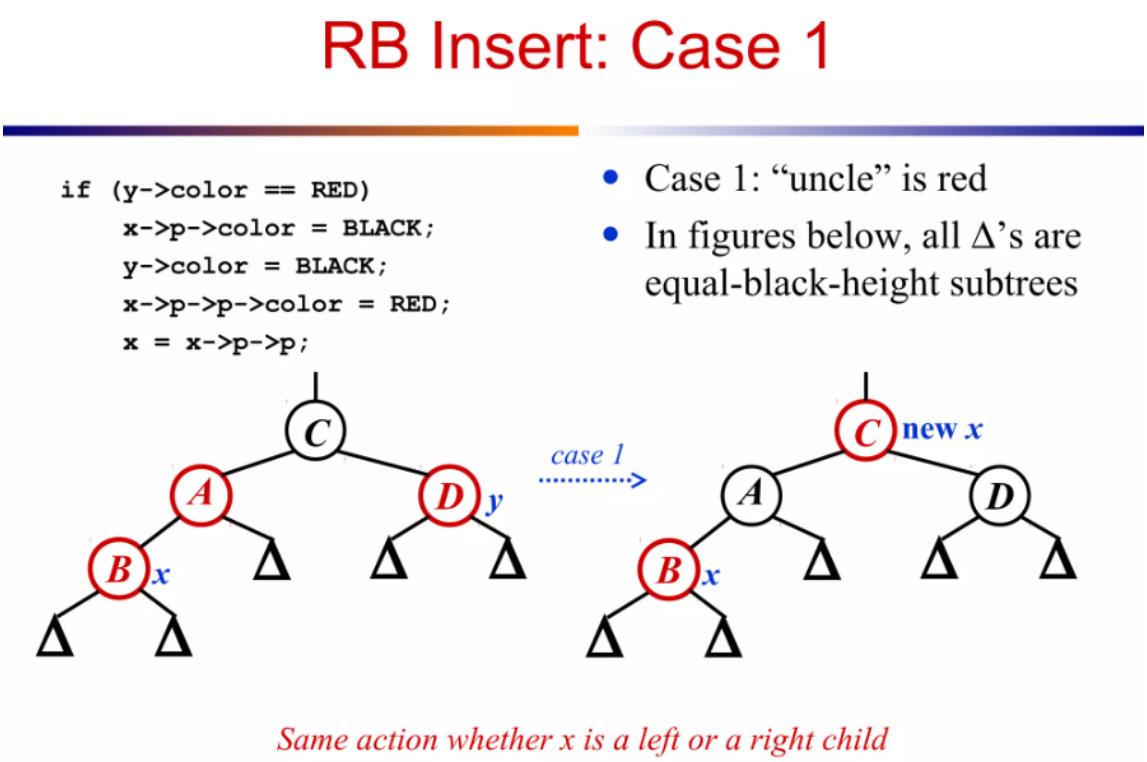
در این حالت، عموی گره x (که تازه درج شده) قرمز رنگ است. این وضعیت ساده‌ترین حالت است و با تغییر رنگ‌های مربوطه حل می‌شود.

۱. گره والد p و عموی y گره x را به رنگ مشکی در می‌آوریم.

۲. گره پدربزرگ p->p را به رنگ قرمز در می‌آوریم.

۳. اشاره‌گر x را به گره پدربزرگ منتقل می‌کنیم تا در تکرار بعدی حلقه (در صورت لزوم) بررسی‌ها از آنجا ادامه پیدا کند.

این عملیات به گونه‌ای انجام می‌شود که تعداد گره‌های مشکی در مسیرهای مختلف درخت حفظ شود.



شکل ۵: درج در درخت قرمز-مشکی: حالت ۱

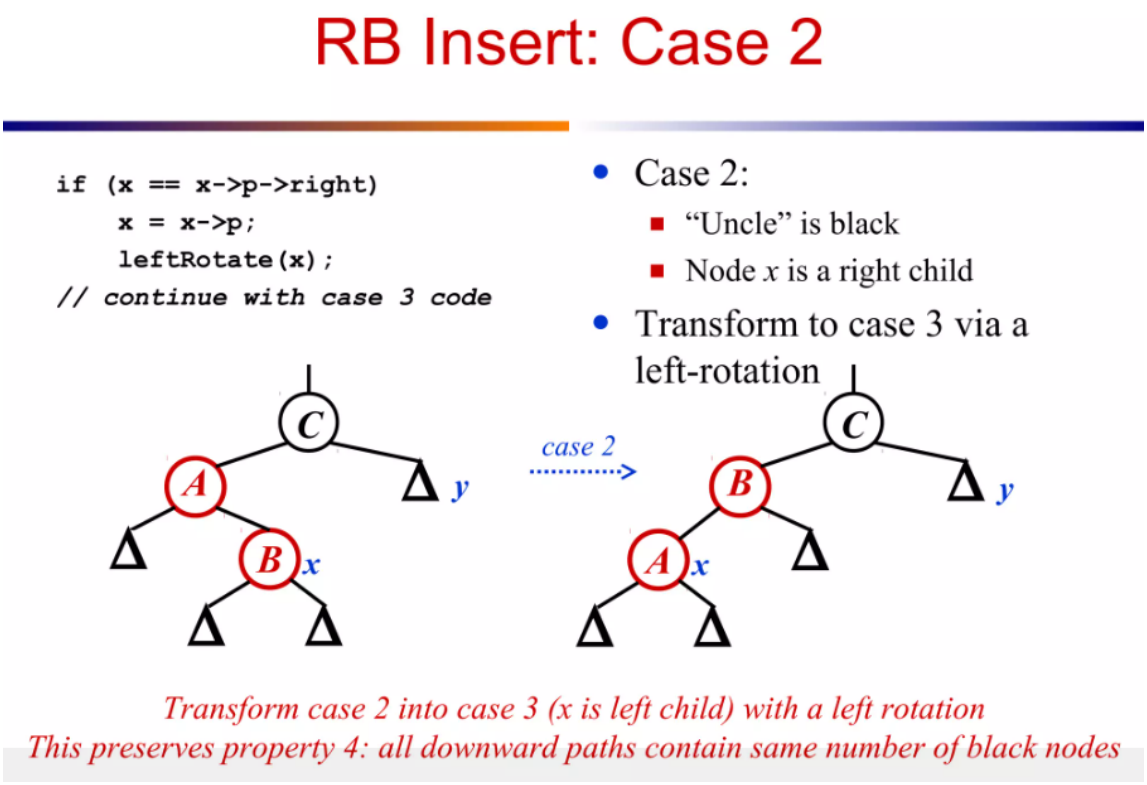
حالت ۲: عموی گره جدید (y) مشکی است و گره x فرزند راست است

در این حالت، عموی گره x مشکی رنگ است و گره x یک فرزند راست برای والد خود p است. برای حل این وضعیت، ابتدا باید این حالت را به حالت ۳ تبدیل کنیم.

۱. یک چرخش به چپ (left rotation) روی گره x انجام می‌دهیم.

۲. پس از چرخش، وضعیت گره‌ها به گونه‌ای می‌شود که می‌توانیم آن را به عنوان حالت ۳ در نظر بگیریم و ادامه عملیات را طبق آن حالت پیش ببریم.

این تبدیل اطمینان حاصل می‌کند که خاصیت ۴ (تمام مسیرهای پایین‌رونده حاوی تعداد یکسانی از گره‌های مشکی هستند) حفظ شود.



شکل ۶: درج در درخت قرمز-مشکی: حالت ۲

در ادامه حالت سوم را بررسی می کنیم :

حالت ۳: عموی گره جدید (y) مشکی است و گره x فرزند چپ است

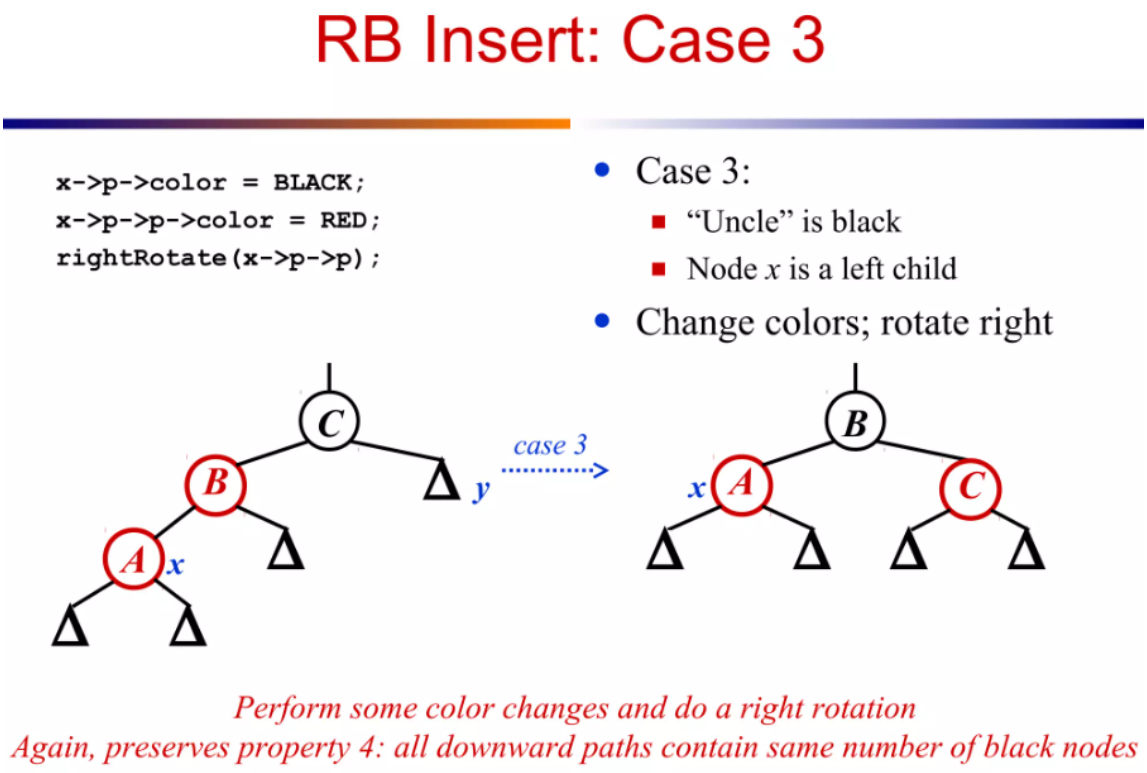
این حالت معمولاً پس از حالت ۲ (در صورت نیاز به تبدیل) یا به صورت مستقیم رخ می‌دهد. در این وضعیت، عموی گره x مشکی است و گره x یک فرزند چپ برای والد خود p است.

۱. رنگ گره والد p را به مشکی تغییر می‌دهیم.

۲. رنگ گره پدربزرگ $p->p$ را به قرمز تغییر می‌دهیم.

۳. یک چرخش به راست (right rotation) روی گره پدربزرگ $p->p$ انجام می‌دهیم.

این عملیات با حفظ خاصیت‌های درخت قرمز-مشکی، تعادل را برقرار می‌کند.
شکل زیر حالت سوم اضافه کردن گره را نشان می دهد :



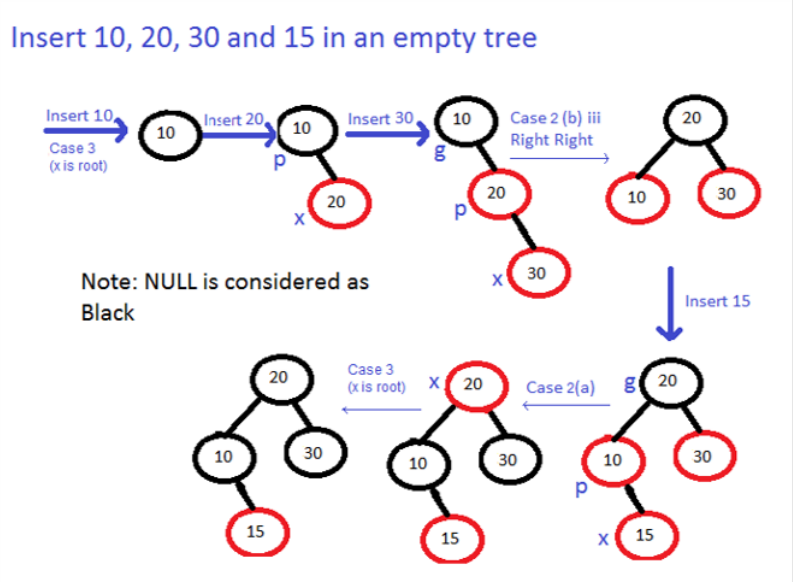
شکل ۷: درج در درخت قرمز-مشکی: حالت ۳

۱.۶ FIXUP درج در درخت قرمز-سیاه

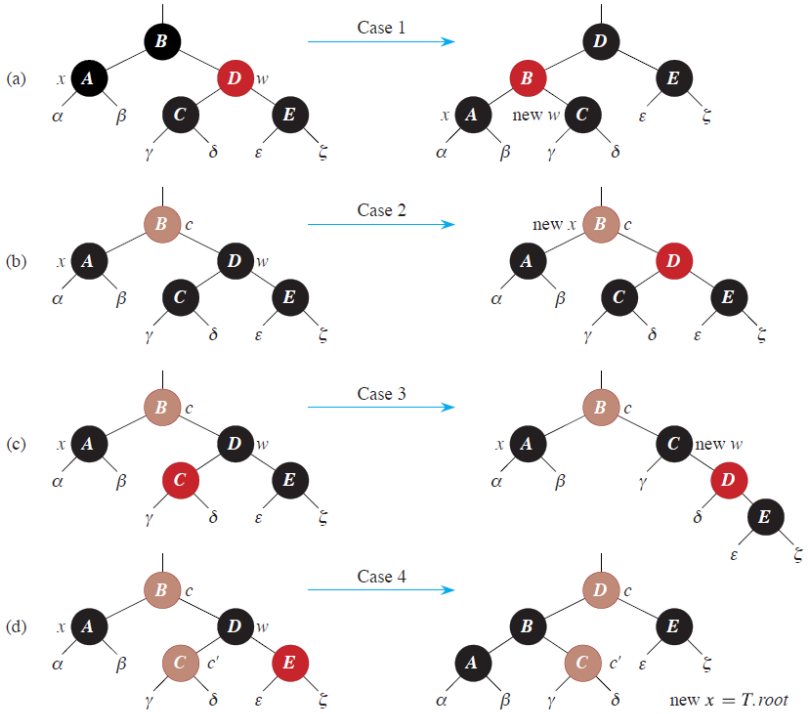
در هنگام درج گره، ممکن است خاصیت‌های درخت قرمز-مشکی نقض شود که با اجرای الگوریتم Fixup و اعمال چرخش و رنگ‌آمیزی، تعادل آن بازگردانده می‌شود.

```
RB-INSERT-FIXUP(T, z):
  while z.p.color == RED
    if z.p == z.p.p.left // is z's parent a left child?
      y = z.p.p.right // is z's uncle
      if y.color == RED // are z's parent and uncle both red?
        z.p.color = BLACK
        y.color = BLACK
        z.p.p.color = RED
        z = z.p.p // case 1
      else // uncle is black
        if z == z.p.right // case 2
          z = z.p
          LEFT-ROTATE(T, z)
        // case 3
        z.p.color = BLACK
        z.p.p.color = RED
        RIGHT-ROTATE(T, z.p.p)
    else // same as lines 3-15, but with "right" and "left" exchanged
      y = z.p.p.left
      if y.color == RED
        z.p.color = BLACK
        y.color = BLACK
        z.p.p.color = RED
        z = z.p.p
      else
        if z == z.p.left
          z = z.p
          RIGHT-ROTATE(T, z)
        z.p.color = BLACK
        z.p.p.color = RED
        LEFT-ROTATE(T, z.p.p)
  T.root.color = BLACK
```

Listing :۴ RB-INSERT-FIXUP



شکل ۸: مثال اجرای Fixup بعد از هر بار insert کردن



شکل ۹: مثال اجرای Fixup بعد از هر بار insert کردن

۷ حذف در درخت قرمز-سیاه

حذف در درخت قرمز-مشکی پیچیده‌تر از درج است، زیرا ممکن است تعادل گره‌های مشکی یا قوانین رنگ نقض شود. الگوریتمRB-DELETEگره را با جانشین مناسب جایگزین کرده و در صورت نیاز، با اجرای RB-DELETE-FIXUP توازن درخت را بازمی‌گرداند. در ادامه، این الگوریتم را بررسی می‌کنیم.

```
RB-DELETE(T, z)
  y = z
  y-original-color = y.color
  if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
       // replace z by its right child
  elseif z.right == T.nil
  x = z.left
  RB-TRANSPLANT(T, z, z.left)
     // replace z by its left child
  else
    y = TREE-MINIMUM(z.right)
       // y is z's successor
    y-original-color = y.color
    x = y.right
    if y   z.right
      RB-TRANSPLANT(T, y, y.right)
         // replace y by its right child
      y.right = z.right
      y.right.p = y
    else
      x.p = y
         // in case x is T.nil
    RB-TRANSPLANT(T, z, y)
       // replace z by y
    y.left = z.left
       // and give z's left child to y
    y.left.p = y
    y.color = z.color
  if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
       // correct any violations
```

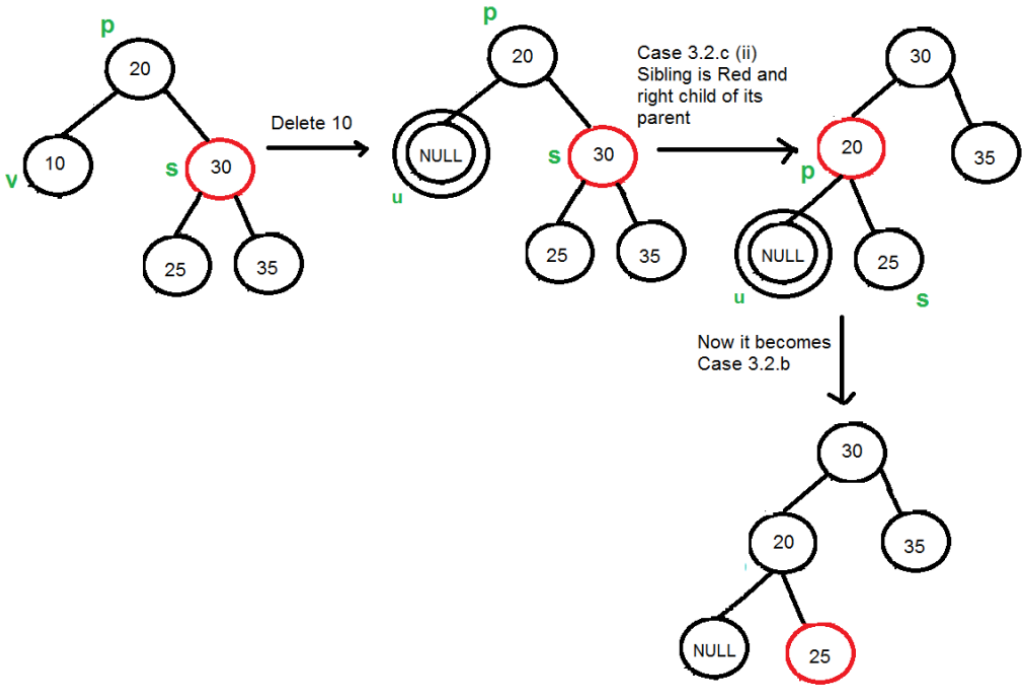
Listing :۵ Red-Black Tree Deletion

RB-DELETE-FIXUP ۱.۷

پس از حذف یک گره در درخت قرمز-مشکی، اگر گره حذف‌شده یا جانشین آن مشکی باشد، ممکن است قوانین قرمز-مشکی نقض شوند.
الگوریتم RB-DELETE-FIXUP برای بازگرداندن این خاصیت‌ها اجرا می‌شود.
این الگوریتم با پیمایش از گره x به سمت ریشه، با چرخش‌ها و تغییر رنگ‌ها تعادل درخت را بازمی‌گرداند.

```
RB-DELETE-FIXUP(T, x)
  while x  T.root and x.color = BLACK
  if x = x.p.left //case1
    w = x.p.right
    if w.color = RED
      w.color = BLACK
      x.p.color = RED
      LEFT-ROTATE(T, x.p)
      w = x.p.right
    if w.left.color = BLACK and
       w.right.color = BLACK  //case2
      w.color = RED
      x = x.p
    else
      if w.right.color = BLACK //case3
        w.left.color = BLACK
        w.color = RED
        RIGHT-ROTATE(T, w)
        w = x.p.right
      w.color = x.p.color // case4
      x.p.color = BLACK
      w.right.color = BLACK
      LEFT-ROTATE(T, x.p)
      x = T.root
  else
    (*@ // same as then-clause with
      "right" and "left" exchanged @*)
```

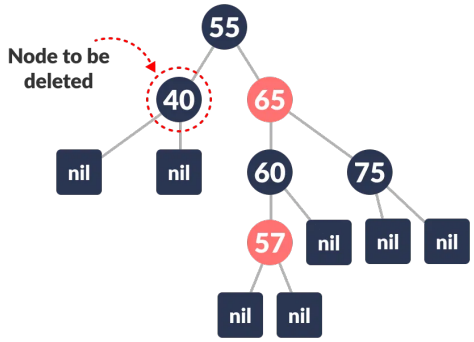
Listing :۶ Red-Black Tree Deletion



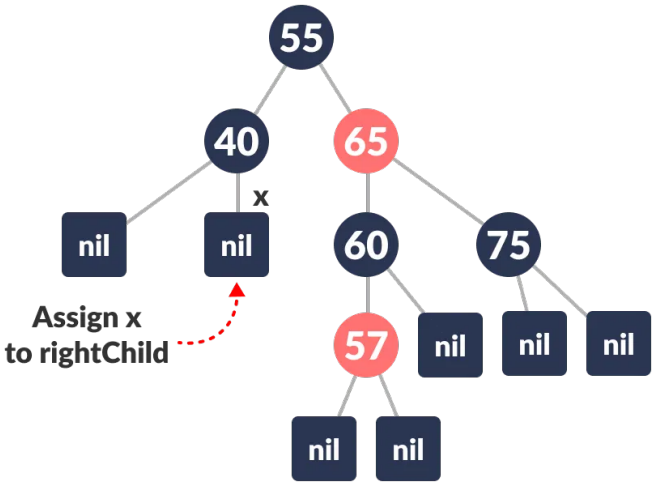
شکل ۱۰: مثال اجرای Fixup بعد از هر بار Delete کردن

۲.۷ مثال پیشرفته RB-DELETE-FIXUP

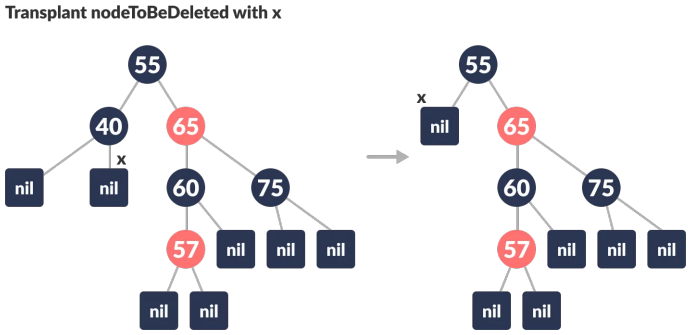
در ادامه یک مثال پیشرفته و چند مرحله ای از حذف گره را تا زمان برقراری کامل تعادل بررسی می کنیم :



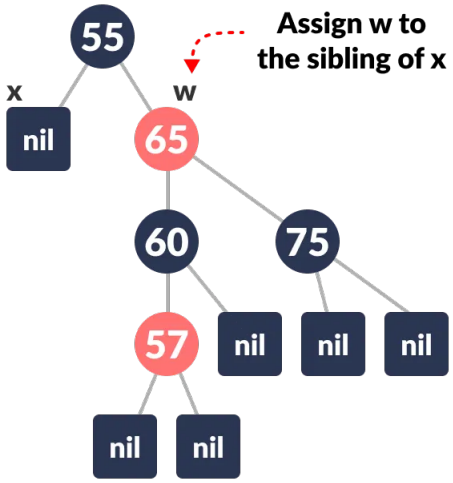
شکل ۱۱: شناسایی گره ۴۰ برای حذف؛ گره ۴۰ یک گره قرمز بدون فرزند است که به سادگی حذف می‌شود.



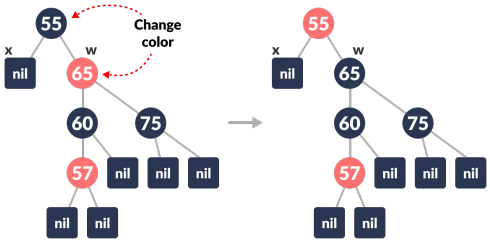
شکل ۱۲: برای حذف ۴۰ ابتدا x را به فرزند راست آن نسبت می‌دهیم.



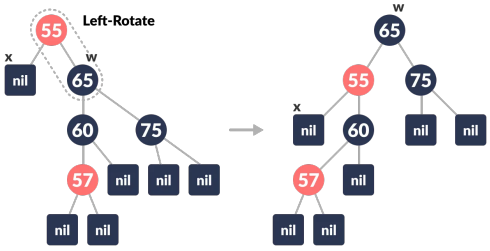
شکل ۱۳: پس از حذف ۴۰، تعداد گره‌های مشکی در مسیرها نابرابر می‌شوند و باید fixup انجام دهیم.



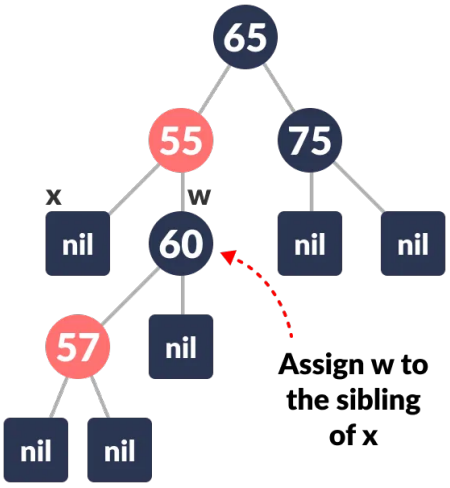
شکل ۱۴: w را به ۶۵ نسبت داده و به عنوان گره برادر/خواهر انتخاب می‌کنیم.



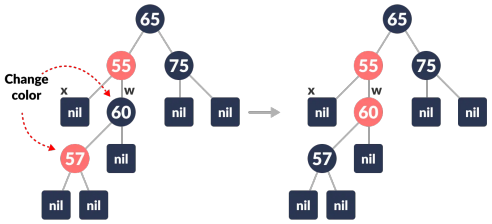
شکل ۱۵: طبق الگوی fixup رنگ ۶۵ و ۵۵ را عوض می‌کنیم.



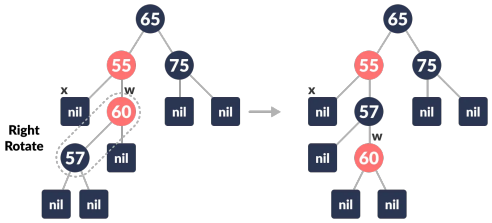
شکل ۱۶: سپس چرخش به چپ حول گره ۵۵ انجام می‌دهیم.



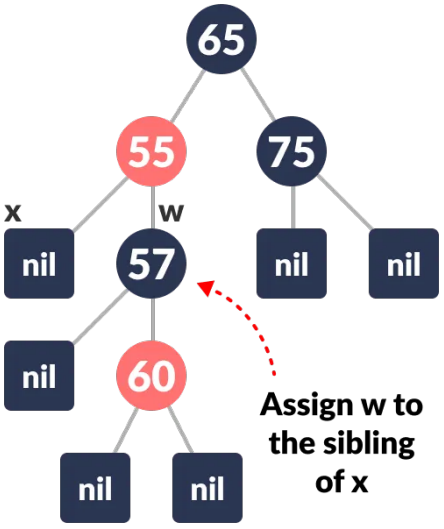
شکل ۱۷: موقعیت فعلی درخت پس از حذف و تعویض. گره ۶۰ به عنوان "w" (برادر) و 'nil' به عنوان "x" (گره دارای دو سیاهی اضافی) مشخص شده است.



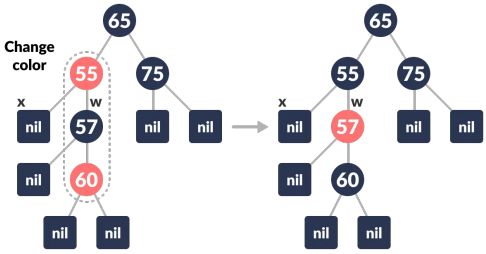
شکل ۱۸: گره ۵۷ از فرزند چپ ۶۰ به فرزند راست ۶۰ منتقل می‌شود و گره ۶۰ رنگ قرمز می‌گیرد.



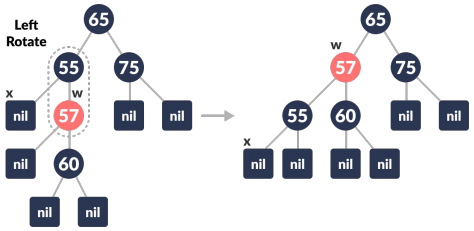
شکل ۱۹: گره ۵۷ جای گره ۶۰ را می‌گیرد و گره ۶۰ به عنوان فرزند راست ۵۷ قرار می‌گیرد. این مرحله شامل تغییر رنگ‌ها برای حفظ خواص قرمز-سیاه است.



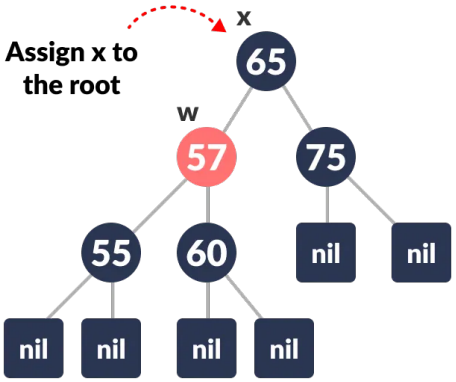
شکل ۲۰: وضعیت پس از حذف گره، که گره ۶۰ به عنوان "x" (نقص دو سیاهی) و گره ۵۷ به عنوان برادر قرمز آن ("sibling") مشخص شده است. (تکرار گام ۷، ولی از زاویه دید x)



شکل ۲۱: چرخش راست در گره ۵۵ انجام می‌شود تا گره ۵۷ به سمت بالا حرکت کند و رنگ‌ها تنظیم شوند.



شکل ۲۲: پس از چرخش و تنظیم رنگ‌ها، گره ۵۷ اکنون ریشه زیردرخت شده است و خواص درخت قرمز-سیاه بازسازی شده‌اند.



شکل ۲۳: شکل نهایی درخت.

۸ منابع

- CLRS (ویرایش چهارم)
- https://www.programiz.com/dsa/deletion-from-a-red-black-tree
- https://medium.com/analytics-vidhya/deletion-in-red-black-rb-tree-92301e1474ea
- https://www.cs.csubak.edu/ msarr/visualizations/RedBlack.html